СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Несмотря на более чем полувековую историю вычислительной техники, рождение теории формальных языков ведет отсчет с 1957 года. В этот год американский ученый Джон Бэкус разработал первый компилятор языка Фортран. Он применил теорию формальных языков, во многом опирающуюся на работы известного ученого-лингвиста Н. Хомского – автора классификации формальных языков. Хомский в основном занимался изучением естественных языков, Бекус применил его теорию для разработки языка программирования. Это дало толчок к разработке сотен языков программирования.

Несмотря на наличие большого количества алгоритмов, позволяющих автоматизировать процесс написания транслятора для формального языка, создание нового языка требует творческого подхода. В основном это относится к синтаксису языка, который, с одной стороны, должен быть удобен в прикладном программировании, а с другой, должен укладываться в область контекстно-свободных языков, для которых существуют развитые методы анализа.

Основы теории формальных языков и практические методы разработки распознавателей формальных языков составляют неотъемлемую часть образования современного инженера-программиста.

Целью данной курсовой работы является:

– освоение основных методов разработки распознавателей формальных языков на примере модельного языка программирования;

– приобретение практических навыков написания транслятора языка программирования;

– закрепление практических навыков самостоятельного решения инженерных задач, умения пользоваться справочной литературой и технической документацией.

1 ПОСТАНОВКА ЗАДАЧИ

Разработать распознаватель модельного языка программирования согласно заданной формальной грамматике.

Распознаватель представляет собой специальный алгоритм, позволяющий вынести решение и принадлежности цепочки символов некоторому языку. Распознаватель можно схематично представить в виде совокупности входной ленты, читающей головки, которая указывает на очередной символ на ленте, устройства управления (УУ) и дополнительной памяти (стек).

Конфигурацией распознавателя является:

- состояние УУ;
- содержимое входной ленты;
- положение читающей головки;
- содержимое дополнительной памяти (стека).

Трансляция исходного текста программы происходит в несколько этапов. Основными этапами являются следующие:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- генерация целевого кода.

Лексический анализ является наиболее простой фазой и выполняется с помощью *регулярной* грамматики. Регулярным грамматикам соответствуют конечные автоматы, следовательно, разработка и написание программы лексического анализатора эквивалентна разработке конечного автомата и его диаграммы состояний (ДС).

Синтаксический анализатор строится на базе *контекстно-свободных* (КС) грамматик. Задача синтаксического анализатора – провести разбор текста программы и сопоставить его с формальным описание языка.

Семантический анализ позволяет учесть особенности языка программирования, которые не могут быть описаны правилами КС-грамматики. К таким особенностям относятся:

- обработка описаний;
- анализ выражений;
- проверка правильности операторов.

Обработки описаний позволяет убедиться в том, что каждая переменная в программе описана и только один раз.

Анализ выражений заключается в том, чтобы проверить описаны ли переменные, участвующие в выражении, и соответствуют ли типы операндов друг другу и типу операции.

Этапы синтаксического и семантического анализа обычно можно объединить.

2 ПОРЯДОК ВЫПОЛНЕНИЯ

1. В соответствии с номером варианта составить описание модельного языка программирования в виде правил вывода формальной грамматики;
2. Составить таблицу лексем и нарисовать диаграмму состояний для распознавания и формирования лексем языка;
3. Разработать процедуру лексического анализа исходного текста программы на языке высокого уровня;
4. Разработать процедуру синтаксического анализа исходного текста методом рекурсивного спуска на языке высокого уровня;
5. Построить программный продукт, читающий текст программы, написанной на модельном языке, в виде консольного приложения;
6. Протестировать работу программного продукта с помощи серии тестов, демонстрирующих все основные особенности модельного языка программирования, включая возможные лексические и синтаксические ошибки.

3 ГРАММАТИКА МОДЕЛЬНОГО ЯЗЫКА

Согласно индивидуальному варианту задания на курсовую работу грамматика языка включает следующие синтаксические конструкции:

<операции_группы_отношения> ::= != | == | < | <= | > | >=

<операции_группы_сложения> ::= + | - | ||

<операции_группы_умножения> ::= * | / | &&

<унарная_операция> ::= !

<программа> ::= *program var* <описание> *begin* <оператор> {; <оператор>} *end*

<описание> ::= <тип> <идентификатор> {, <идентификатор> }

<тип> ::= *int* | *float* | *bool*

<оператор> ::= <составной> | <присваивания> | <условный> | <фиксированного_цикла> | <цикла> | <ввода> | <вывода>

<составной> ::= *begin* <оператор> {; <оператор> } *end*

<присваивания> ::= <идентификатор> := <выражение>

<условный> ::= *if* «(» <выражение>«)» then <оператор> [*else* <оператор>]

<фиксированного_цикла> ::= *for* <присваивания> *to* (<выражение>) [*step* <выражение>] <оператор> *next*

<цикла> ::= *while* «(»<выражение> «)»<оператор>

<ввода> ::= *readln*(<идентификатор>) {, <идентификатор> }

<вывода> ::= *writeln*(<выражение>){, <выражение> }

<выражение> ::= <операнд> {<операции_группы_отношения> <операнд>}

<операнд> ::= <слагаемое> {<операции_группы_сложения> <слагаемое>}

<слагаемое> ::= <множитель> {<операции_группы_умножения> <множитель>}

<множитель> ::= <идентификатор> | <число> | <логическая_константа> | <унарная_операция> <множитель> | (<выражение>)

<логическая_константа> ::= true | false

<идентификатор> ::= <буква>{<буква> | <цифра>}

<число> ::= <цифра>{<цифра> | *a* | A | *b* | B | c | C | d |D | e | E | f | F | . | + | -}

<буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Здесь для записи правил грамматики используется форма Бэкуса-Наура (БНФ). В записи БНФ левая и правая части порождения разделяются символом “::=”, нетерминалы заключены в угловые скобки, а терминалы – просто символы, используемые в языке. Жирным выделены терминалы, представляющие собой ключевые слова языка.

4 РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

Лексический анализатор – подпрограмма, которая принимает на вход исходный текст программы и выдает последовательность *лексем* – минимальных элементов программы, несущих смысловую нагрузку.

В модельном языке программирования выделяют следующие типы лексем:

- ключевые слова;
- ограничители;
- числа;
- идентификаторы.

При разработке лексического анализатора, ключевые слова и ограничителя известны заранее, идентификаторы и числовые константы – вычисляются в момент разбора исходного текста.

Для каждого типа лексем предусмотрена отдельная таблица. Таким образом, внутреннее представление лексемы – пара чисел (n, k), где n – номер таблицы лексем, k – номер лексемы в таблице.

Кроме того, в исходном коде программы кроме ключевых слов, идентификаторов и числовых констант может находиться произвольное число пробельных символов («пробел», «табуляция», «перенос строки», «возврат каретки») и комментариев, заключенных в фигурные скобки.

Лексический анализ текста проводится по регулярной грамматике. Известно, что регулярная грамматика эквивалентна конченому автомату, следовательно, для написания лексического анализатора необходимо построить диаграмму состояний, соответствующего конечного автомата (рис. 1).

Исходные код лексического анализатора приведен в Приложении А.

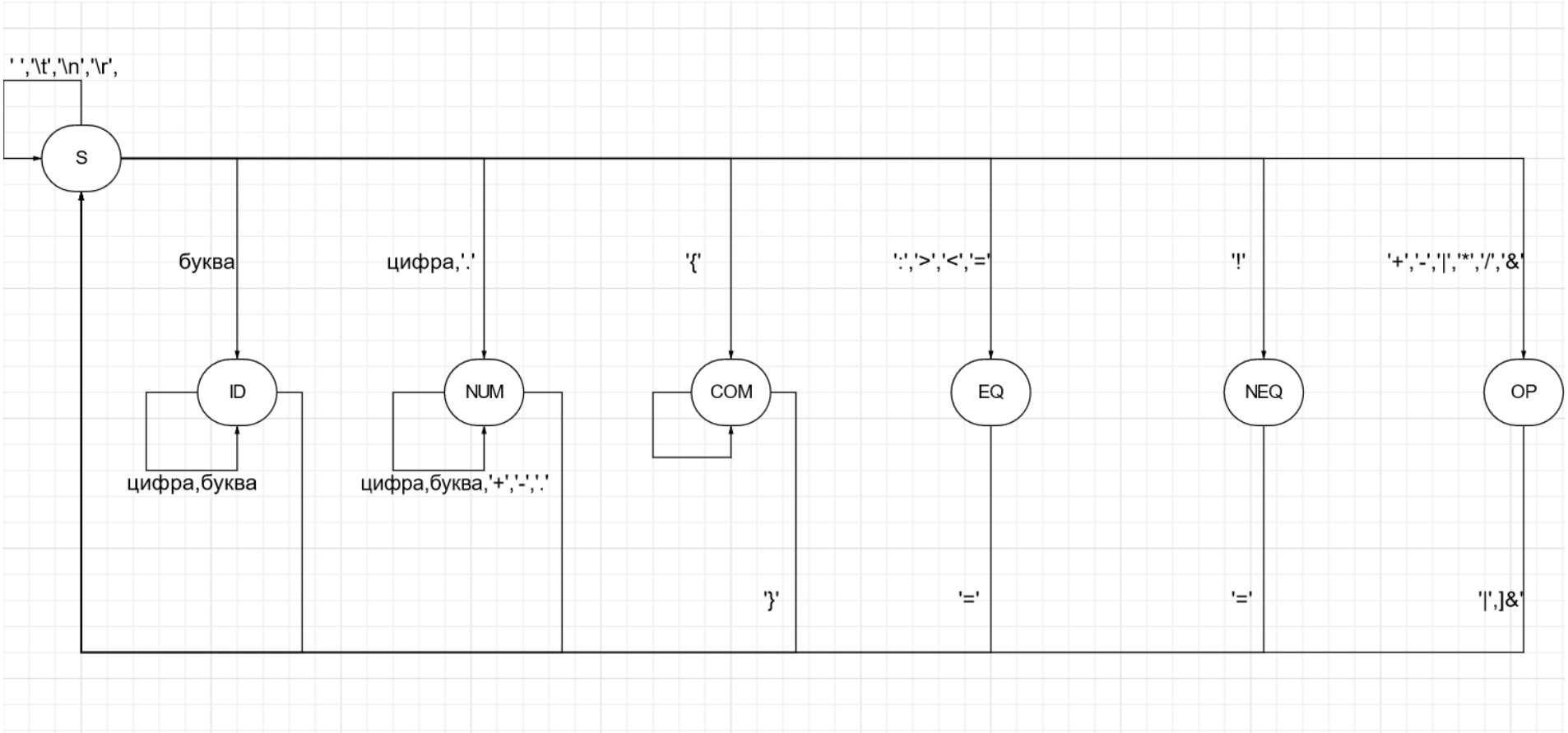

Рисунок 1 – Диаграмма состояний лексического анализатор

5 РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Будем считать, что лексический и синтаксической анализаторы взаимодействуют следующим образом. Если синтаксическому анализатору для анализа требуется очередная лексема, он запрашивает ее у лексического анализатора. Таким образом, разбор исходного текста программы идет под управлением подпрограммы синтаксического анализатора (*parser*).

Разработку синтаксического анализатора проведем с помощью метода рекурсивного спуска (РС). В основе метода лежит тот факт, что каждому нетерминалу ставится в соответствие рекурсивная функция. Для того, чтобы в явном виде представить множество рекурсивных функций, перепишем грамматические правила следующим образом:

S → ***program*** V ***begin*** B ***end***

V → var T

T → [int | float | bool] I {,I}

B → ***begin*** B {; B} ***end***

B → I := W

B → ***for*** I := W ***to*** W **step** W B **next**

B → ***if (*** W ***)*** B ***else*** B

B → ***while (*** W ***)*** B

B → ***readln*** I {,I}

B → ***writeln*** W {,W}

W → O{[== | > | < | >= | <= |! =] O}

O → S{[+ | − | ||] S}

S → M{[* | / | &&] M}

M → I | N | L| ! M | (W)

$L \rightarrow \boldsymbol{true} \mid \boldsymbol{false}$

$I \rightarrow C \mid IC \mid IR$

$N \rightarrow R \mid NR$

$C \rightarrow a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z$

$R \rightarrow 0 \mid 1 \mid \ldots \mid 9 \mid a \mid A \mid b \mid B \mid c \mid C \mid d \mid D \mid e \mid E \mid f \mid F \mid . \mid + \mid -$

Здесь правила для нетерминалов L, I, N, C и R описаны на этапе лексического разбора. Следовательно, остается описать функции для нетерминалов P, D1, D, B, S, E, E1, T, F.

Исходный код синтаксического анализатора приведен в Приложении Б.

6 СЕМАНТИЧЕСКИЙ АНАЛИЗ

Некоторые особенности модельного языка не могут быть описаны контекстно-свободной грамматикой. К таким правилам относятся:

- любой идентификатор, используемый в теле программы должен быть описан;
- повторное описание одного и того же идентификатора не разрешается;
- в операторе присваивания типы идентификаторов должны совпадать;
- в условном операторе и операторе цикла в качестве условия допустимы только логические выражения;

операнды операций отношения должны быть целочисленными.

Указанные особенности языка разбираются на этапе семантического анализа. Удобно процедуры семантического анализа совместить с процедурами синтаксического анализа. На практике это означает, что в рекурсивные функции встраиваются дополнительные контекстно-зависимые проверки.

В данном варианте все идентификаторы объявляются только в начале программы, и все идентификаторы имеют один тип, что существенно облегчает семантический анализ.

7 ТЕСТИРОВАНИЕ ПРОГРАММЫ

В качестве программного продукта разработано консольное приложение, Приложение принимает на вход исходный текст программы на модельном языке и выдает в качестве результата сообщение о синтаксической и семантической корректности написанной программы. В случае обнаружения ошибки программа выдает сообщение об ошибке с некорректной лексемой и её координатами в тексте. Рассмотрим примеры.

1. Исходный код программы приведен в листинге 1.

Листинг 1 – Тестовая программа

```
program var
      int a, b, alpha, beta
begin
      b := 77d;
      readln alpha, beta;
      for a := 0d to b + 2d step 10b + 36o
      begin
            if (alpha >= 12D)
            begin
                  beta := .5e+12;
                  beta := alpha + beta
            end
            else
                  alpha := 50e+12
      end
      next {Комментарий}
      while (alpha)
            for a := 0d to b + 2d
                  a := a + 2d
            next
      writeln beta + alpha * b / a;
end
```

Данная программа синтаксически корректна и использует все возможные операторы, поэтому анализатор выдает следующее сообщение (рис. 2).

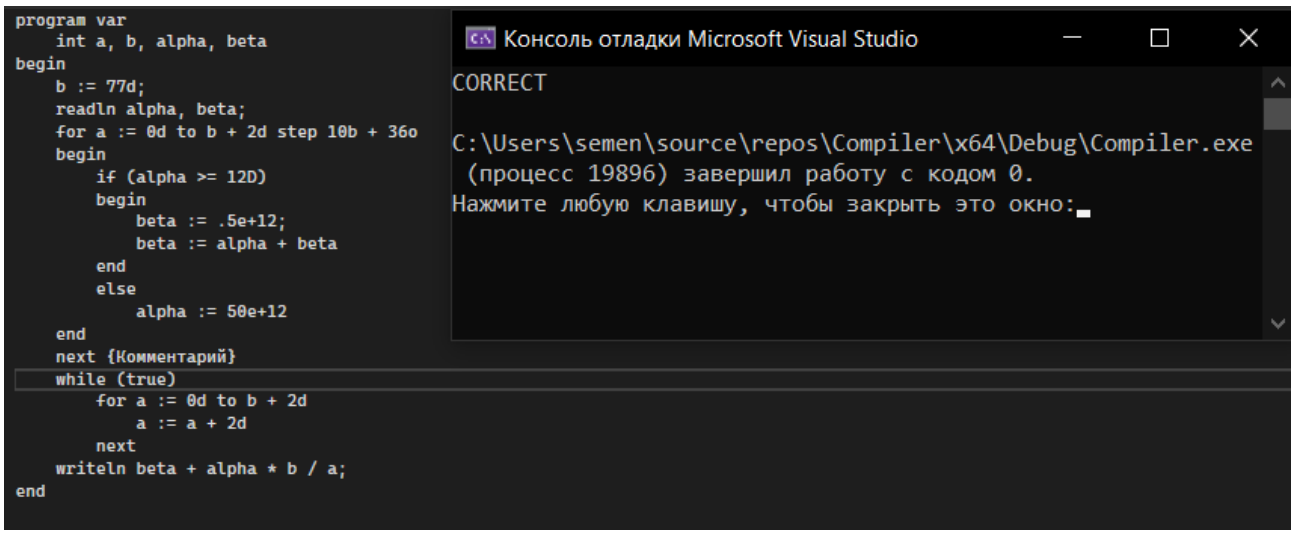

Рисунок 2 – Пример синтаксически корректной программы

2. Исходный код программы, содержащий семантическую ошибку, приведен на рис. 3 совместно с сообщением об ошибке.

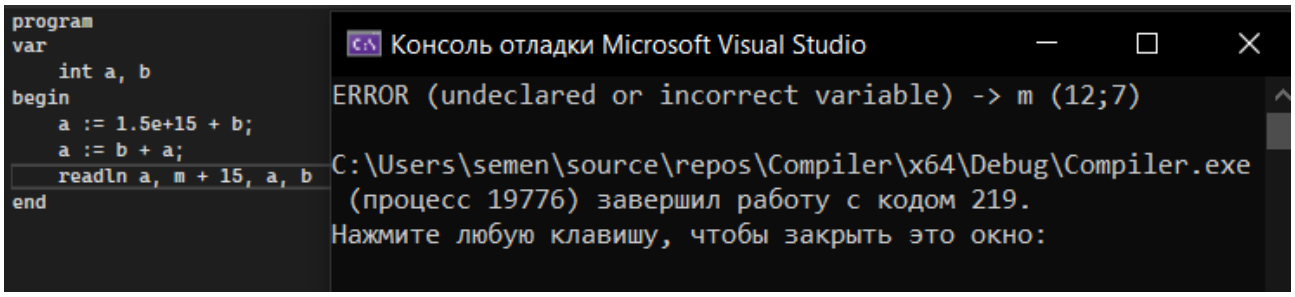

Рисунок 3 – Пример семантически некорректной программы

Здесь ошибка допущена в строке 7: использование необъявленной переменной m. В сообщении об ошибке указана сама ошибка, строка и номер первого вхождения символа, и программа завершается с кодом 219, что соответствует этой ошибке.

3. Исходный код программы, содержащий синтаксическую ошибку, приведен на рис. 4 совместно с сообщением об ошибке.

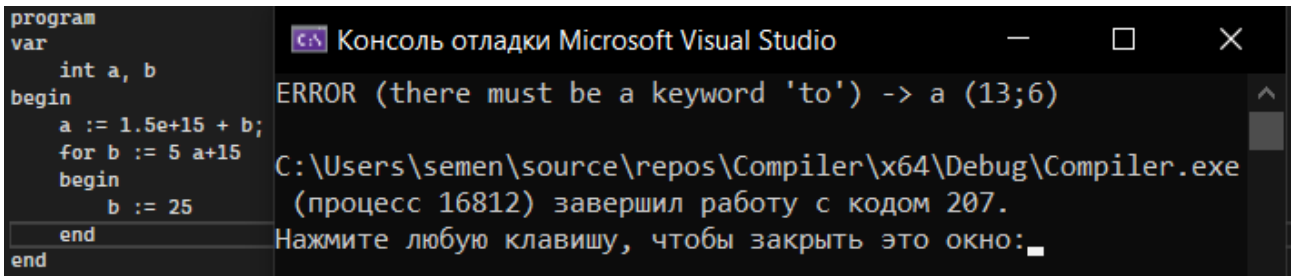

Рисунок 4 – Пример синтаксически некорректной программы

Здесь ошибка допущена в строке 6: отсутствует ключевое слова «to», которое необходимо в операторе фиксированного цикла. В сообщении об ошибке указана сама ошибка, строка и номер первого вхождения символа, и программа завершается с кодом 207, что соответствует этой ошибке.

ЗАКЛЮЧЕНИЕ

В работе представлены результаты разработки анализатора языка программирования. Грамматика языка задана с помощью правил вывода и описана в форме Бэкуса-Наура (БНФ). Согласно грамматике, в языке присутствуют лексемы следующих базовых типов: числовые константы, переменные, разделители и ключевые слова.

Разработан лексический анализатор, позволяющий разделить последовательность символов исходного текста программы на последовательность лексем. Лексический анализатор реализован на языке высокого уровня C++ в виде структуры lex.

Разбором исходного текста программы занимается синтаксический анализатор, который реализован в виде структуры parser на языке C++. Анализатор распознает входной язык по методу рекурсивного спуска. Для применимости необходимо было преобразовать грамматику, в частности, специальным образом обрабатывать встречающиеся итеративные синтаксически конструкции (нетерминалы V, T, B, O и S).

В код рекурсивных функций включены проверки дополнительных семантических условий.

Тестирование программного продукта показало, что синтаксически и семантически корректно написанная программа успешно распознается анализатором, а программа, содержащая ошибки, отвергается.

В ходе работы изучены основные принципы построения интеллектуальных систем на основе теории автоматов и формальных грамматик, приобретены навыки лексического, синтаксического и семантического анализа предложений языков программирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Свердлов С. З. Языки программирования и методы трансляции: учебное пособие. – Санкт-Петербург: Лань, 2019.

2. Малявко А. А. Формальные языки и компиляторы: учебное пособие для вузов. – М.: Юрайт, 2020.

3. Унгер А.Ю. Основы теории трансляции: учебник. – М.: МИРЭА – Российский технологический университет, 2022.

4. Антик М. И., Казанцева Л. В. Теория формальных языков в проектировании трансляторов: учебное пособие. – М.: МИРЭА, 2020.

5. Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий. – М.: Вильямс, 2008.

ПРИЛОЖЕНИЯ

Приложение А – структура лексического анализатора

Приложение Б – структура синтаксического анализатора

Приложение А

Структура лексического анализатора

Листинг А.1 – структура lex

```
struct lex
{
	std::vector<token> tokensVec;	// Поле хранящее все токены
	std::string inputString = "";	// Поле хранящее считываемую строку
	size_t x_coord = 0;		// Поле хранящее номер считываемой строки
	size_t y_coord = 0;		// Поле хранящее номер считываемого символа
	char c = 0;			// Поле хранящее анализируемый символ
	std::string lexBuff = "";	// Поле хранящее анализируемую лексему
	state cState = S;		// Поле хранящее состояние конечного
автомата
	token buffToken;		// Поле хранящее анализируемый токен
	// Входной буфер
	lex(const char* fileName)
	{
		std::ifstream inputFile(fileName);
		while (getline(inputFile,inputString))
		{
			y_coord++;
			reading_mechanism(inputString);
		}
	}
	// Считывающая машина представляющая собой конечный автомат
	void reading_mechanism(std::string inputString)
	{
		for (size_t i = 0; i < inputString.size(); i++)
		{
			c = inputString[i];
			switch (cState)
			{
			case S:
				if (c == ' ' || c == '\n' || c == '\t' || c == '\r')
					cState = S;
				else if (isalpha(c))
				{
					clear_string();
					x_coord = i + 1;
					add_to_string();
					cState = ID;
				}
				else if (isdigit(c) || c =='.')
				{
					clear_string();
					x_coord = i + 1;
					add_to_string();
					cState = NUM;
				}
				else if (c == '{')
				{
					clear_string();
					x_coord = i + 1;
					add_to_string();
					cState = COM;
				}
				else if (c == ':' || c == '>' || c == '<' || c == '=')
```

Продолжение листинга А.1

```
{
                              clear_string();
                              x_coord = i + 1;
                              add_to_string();
                              cState = EQ;
                        }
                        else if (c == '!')
                        {
                              clear_string();
                              x_coord = i + 1;
                              cState = NEQ;
                              add_to_string();
                        }
                        else if (c == '+' || c == '-' || c == '|' || c == '*'
|| c == '/' || c == '&')
                        {
                              clear_string();
                              x_coord = i + 1;
                              add_to_string();
                              cState = OP;
                        }
                        else
                        {
                              if (key_lex() || sep_lex())
                              {
                                    clear_string();
                                    x_coord = i + 1;
                                    add_to_string();
                              }
                              else
                              {
                                    std::cerr << "ERROR (inncorrect symbol) ->
" << c << " (" << x_coord << ',' << i++ << ')' << std::endl;
                                    exit(101);
                              }
                        }
                        break;
                  case NUM:
                        if (((c == '+' || c == '-') && tolower(inputString[i-
1]) == 'e') || isalpha(c) || isdigit(c) || c == '.')
                              add_to_string();
                        else
                        {
                              i--;
                              cState = S;
                        }
                        break;
                  case ID:
                        if (isalpha(c) || isdigit(c))
                              add_to_string();
                        else{
                              i--;
                              cState = S;
                        }
                        break;
                  case COM:
                        if (c == '}'){
                              add_to_string();
                              lexBuff = "";
                              cState = S;
```

Продолжение листинга А.1

```
                        }
                        else
                              add_to_string();
                        break;
                  case EQ:
                        if (c == '=')
                        {
                              add_to_string();
                              cState = S;
                        }
                        else if (sep_lex())
                        {
                              std::cerr   <<   "ERROR   (incorrect   comparison
opperator)  ->  "  <<  lexBuff  <<  "  ("  <<  y_coord  <<  ','  <<  x_coord  <<  ')'  <<
std::endl;
                              exit(103);
                        }
                        else if (lexBuff[0] != c)
                        {
                              std::cerr   <<   "ERROR   (incorrect   assigment
opperator)  ->  "  <<  lexBuff  <<  "  ("  <<  y_coord  <<  ','  <<  x_coord  <<  ')'  <<
std::endl;
                              exit(104);
                        }
                        else
                        {
                              i--;
                              cState = S;
                        }
                        break;
                  case NEQ:
                        if (c == '=')
                        {
                              add_to_string();
                              cState = S;
                        }
                        else
                        {
                              i--;
                              cState = S;
                        }
                        break;
                  case OP:
                        if ((c == '|' || c == '&') && lexBuff[0] == c)
                        {
                              add_to_string();
                              cState = S;
                        }
                        else if ((c == '|' || c == '&') && lexBuff[0] != c)
                        {
                              std::cerr << "ERROR (incorrect logical opperator)
-> " << lexBuff << " (" << y_coord << ',' << x_coord << ')' << std::endl;
                              exit(105);
                        }
                        else{
                              i--;
                              cState = S;
                        }
                        break;
                  default:
                        break;}}
```

Продолжение листинга А.1

```
clear_string();
            proc_tokens();
      }
      // Метода добавления символа в лексему
      void add_to_string() {lexBuff += c;}
      // Метод анализа лексемы
      void clear_string()
      {
            buffToken.value = lexBuff;
            buffToken.x_coord = x_coord;
            buffToken.y_coord = y_coord;
            lexBuff = "";
            if (key_lex() || sep_lex() || num_lex() || id_lex())
                  tokensVec.push_back(buffToken);
            else
            {
                  std::cerr << "ERROR (incorrect identificator) -> " <<
buffToken.value << " (" << y_coord << ',' << x_coord << ')' << std::endl;
                  exit(102);
            }
      }
      // Метод проверки на ключевые слова
      bool key_lex()
      {
            int i = 0;
            while (key_words[i] != NULL)
            {
                  if (buffToken.value == key_words[i])
                  {
                        buffToken.type = type_key_words[i];
                        return true;
                  }
                  i++;
            }
            return false;
      }
      // Метод проверки на разделительные слова
      bool sep_lex()
      {
            int i = 0;
            while (sep_words[i] != NULL)
            {
                  if (buffToken.value == sep_words[i]){
                        buffToken.type = type_sep_words[i];
                        return true;
                  }
                  i++;
            }
            return false;
      }
      // Метод проверки на число
      bool num_lex()
      {
            char lastSymb = tolower(buffToken.value[buffToken.value.size() -
1]);
            if ((isdigit(buffToken.value[0]) &&
                  (lastSymb == 'b' || lastSymb == 'o' || lastSymb == 'd' ||
lastSymb == 'h'))
                  || check_e(buffToken.value))
            {
                  buffToken.type = LEX_NUM;
```

Продолжение листинга А.1

```
return true;
            }
            return false;
      }
      // Метод проверки на идентификаторв
      bool id_lex()
      {
            if (isalpha(buffToken.value[0]))
            {
                  buffToken.type = LEX_ID;
                  return true;
            }
            return false;
      }
      // МЕтод проверки действительного числа
      bool check_e(std::string buffStr)
      {
            size_t ePlace = -1;
            size_t dPlace = -1;
            if (buffStr[buffStr.size() - 1] == '.' || buffStr[buffStr.size() -
1] == '+' || buffStr[buffStr.size() - 1] == '-')
                  return false;
            for (size_t i = 0; i < buffStr.size(); i++)
            {
                  if (tolower(buffStr[i]) == 'e' && ePlace == -1)
                  {
                        if (dPlace != -1 && dPlace == i - 1)
                              return false;
                        ePlace = i;
                  }
                  else if (buffStr[i] == '.' && dPlace == -1)
                  {
                        if (ePlace != -1)
                              return false;
                        dPlace = i;
                  }
                  else if ((dPlace == i - 1 && (isalpha(buffStr[i]))) ||
((buffStr[i] == '+' || buffStr[i] == '-') && ePlace != i - 1)
                        || (buffStr[i] == '.' && ((dPlace != -1 && dPlace != i)
|| ePlace != -1)) || (tolower(buffStr[i]) == 'e' && ((ePlace != -1 && ePlace !=
i))))
                  {
                        return false;
                  }
            }
            if (ePlace != -1 && buffStr[ePlace+1] != '+' && buffStr[ePlace+1]
!= '-')
                  return false;
            return true;
      }
      // Метод удаления пустых лексем
      void proc_tokens()
      {
            for (auto el = tokensVec.begin(); el != tokensVec.end(); )
                  if (el->type == 0) el = tokensVec.erase(el);
                  else el++;
      }
};
```

Листинг А.1 – структура lex

```
struct lex
{
	std::vector<token> tokensVec;	// Поле хранящее все токены
	std::string inputString = "";	// Поле хранящее считываемую строку
	size_t x_coord = 0;		// Поле хранящее номер считываемой строки
	size_t y_coord = 0;		// Поле хранящее номер считываемого символа
	char c = 0;			// Поле хранящее анализируемый символ
	std::string lexBuff = "";	// Поле хранящее анализируемую лексему
	state cState = S;		// Поле хранящее состояние конечного
автомата
	token buffToken;		// Поле хранящее анализируемый токен
	// Входной буфер
	lex(const char* fileName)
	{
		std::ifstream inputFile(fileName);
		while (getline(inputFile,inputString))
		{
			y_coord++;
			reading_mechanism(inputString);
		}
	}
	// Считывающая машина представляющая собой конечный автомат
	void reading_mechanism(std::string inputString)
	{
		for (size_t i = 0; i < inputString.size(); i++)
		{
			c = inputString[i];
			switch (cState)
			{
			case S:
				if (c == ' ' || c == '\n' || c == '\t' || c == '\r')
					cState = S;
				else if (isalpha(c))
				{
					clear_string();
					x_coord = i + 1;
					add_to_string();
					cState = ID;
				}
				else if (isdigit(c) || c =='.')
				{
					clear_string();
					x_coord = i + 1;
					add_to_string();
					cState = NUM;
				}
				else if (c == '{')
				{
					clear_string();
					x_coord = i + 1;
					add_to_string();
					cState = COM;
				}
				else if (c == ':' || c == '>' || c == '<' || c == '=')
```

Приложение Б

Структура синтаксического анализатора

Листинг Б.1 – структура parser

```
struct parser
{
	size_t idx = 0;// Поле хранящее индекс обрабатываемого токена
	size_t tokenCount = 0;		// Поле хранящее количество всех токенов
	std::vector <token> tokensVec;// Поле хранящее все токены
	std::vector <varTokens> tokensVar;  // Поле хранящее все переменные
	varTokens singleTokenVar;	// Поле хранящее обрабатываемую переменную

	int ifWhileCheck = 0;
	bool ifWhileFlag = false;

	// Конструктор парсера
	parser(std::vector <token> tokensVec)
	{
		idx = 0;
		if (tokensVec[idx].type != LEX_PROGRAM)
		{
			std::cerr << "ERROR (there must be a keyword 'program') -> "
<< tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
			exit(201);
		}
		this->tokensVec = tokensVec;
		this->tokenCount = tokensVec.size();
		idx++;
	}
	// Запуск парсера
	void start_prog()
	{
		if (tokensVec[idx].type == LEX_VAR)
		{
			idx++;
			start_vars();
		}
		if (tokensVec[idx].type == LEX_BEGIN)
		{
			start_begin();
		}
		else
		{
			std::cerr << "ERROR (the program structure is broken) -> " <<
tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
			exit(202);
		}
	}
// Метод обработки головы программы
	void start_vars(){
		switch (tokensVec[idx].type)
		{
		case LEX_INT:
			singleTokenVar.varType = "INT";
			idx++;
			break;
		case LEX_FLOAT:
			singleTokenVar.varType = "FLOAT";
```

Продолжение листинга Б.1

```
                  idx++;
                  break;
            case LEX_BOOL:
                  singleTokenVar.varType = "BOOL";
                  idx++;
                  break;
            default:
                  std::cerr << "ERROR (there must be a type of variable) -> "
<< tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                  exit(203);
            }
            start_id();
            tokensVar.push_back(singleTokenVar);
            while (tokensVec[idx].type == LEX_COMMA)
            {
                  idx++;
                  for (int i = 0; i < tokensVar.size(); i++)
                  {
                        if (tokensVar[i].varName == tokensVec[idx].value)
                        {
                              std::cerr << "ERROR (re-declaring a variable) ->
" << tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                              exit(222);
                        }
                  }
                  start_id();
                  tokensVar.push_back(singleTokenVar);
            }
      }
      // Метод проверки корректности переменной
      void start_id(bool flagToken = true)
      {
            if (tokensVec[idx].type == LEX_ID)
            {
                  if (flagToken)
                        singleTokenVar.varName = tokensVec[idx].value;
                  else
                        id_check();
                  idx++;
            }
            else
            {
                  std::cerr << "ERROR (there must be an id) -> " <<
tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                  exit(204);
            }
      }
// Метод обработки тела программы
      void start_begin()
      {
            switch (tokensVec[idx].type)
            {
            case LEX_ID:
                  id_check();
                  idx++;
                  if (tokensVec[idx].type == LEX_ASSIGN){
                        idx++;
                        start_V();}
```

Продолжение листинга Б.1

```
else
                  {
                        std::cerr << "ERROR (there must be an assignment sign)
-> " << tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                        exit(205);
                  }
                  break;
            case LEX_FOR:
                  idx++;
                  if (tokensVec[idx].type == LEX_ID)
                  {
                        id_check();
                        idx++;
                        if (tokensVec[idx].type == LEX_ASSIGN)
                        {
                              idx++;
                              start_V();
                              if (tokensVec[idx].type == LEX_TO)
                              {
                                    idx++;
                                    start_V();
                                    if (tokensVec[idx].type == LEX_STEP)
                                    {
                                          idx++;
                                          start_V();
                                    }
                                    start_begin();
                                    if (tokensVec[idx].type == LEX_NEXT)
                                    {
                                          idx++;
                                          start_begin();
                                    }
                                    else
                                    {
                                          std::cerr << "ERROR (there must be a
keyword 'next') -> " << tokensVec[idx].value << " (" << tokensVec[idx].x_coord
<< ';' << tokensVec[idx].y_coord << ')' << std::endl;
                                          exit(206);
                                    }
                              }
                              else
                              {
                                    std::cerr << "ERROR (there must be a keyword
'to') -> " << tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                                    exit(207);
                              }
                        }else{
                              std::cerr << "ERROR (there must be an assignment
sign) -> " << tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                              exit(208);
                        }
                  }else {
                        std::cerr << "ERROR (there must be an ID) -> " <<
tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                        exit(209);
                  }
                  break;
```

Продолжение листинга Б.1

```
case LEX_IF:
                  idx++;
                  if (tokensVec[idx].type == LEX_LPAREN)
                  {
                        ifWhileCheck = idx - 1;
                        idx++;
                        ifWhileFlag = false;
                        start_V();
                        if (ifWhileFlag == false)
                        {
                              std::cerr << "ERROR ('if' statement accepts only
logical expressions) -> " << tokensVec[ifWhileCheck].value << " (" <<
tokensVec[ifWhileCheck].x_coord << ';' << tokensVec[ifWhileCheck].y_coord <<
')' << std::endl;
                              exit(220);
                        }
                        if (tokensVec[idx].type == LEX_RPAREN)
                        {
                              idx++;
                              start_begin();
                              if (tokensVec[idx].type == LEX_ELSE)
                              {
                                    idx++;
                                    start_begin();
                              }
                        }
                        else
                        {
                              std::cerr << "ERROR (there must be an ')') -> "
<< tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                              exit(210);
                        }
                  }
                  else
                  {
                        std::cerr << "ERROR (there must be an '(') -> " <<
tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                        exit(211);
                  }
                  break;
case LEX_WHILE:
                  idx++;
                  if (tokensVec[idx].type == LEX_LPAREN)
                  {
                        ifWhileCheck = idx - 1;
                        idx++;
                        ifWhileFlag = false;
                        start_V();
                        if (ifWhileFlag == false)
                        {
                              std::cerr << "ERROR ('while' statement accepts
only logical expressions) -> " << tokensVec[ifWhileCheck].value << " (" <<
tokensVec[ifWhileCheck].x_coord << ';' << tokensVec[ifWhileCheck].y_coord <<
')' << std::endl;
                              exit(221);
                        }
if (tokensVec[idx].type == LEX_RPAREN){
                              idx++;
                              start_begin();}
```

Продолжение листинга Б.1

```
else
                        {
                              std::cerr << "ERROR (there must be an ')') -> "
<<  tokensVec[idx].value  <<  "  ("  <<  tokensVec[idx].x_coord  <<  ';'  <<
tokensVec[idx].y_coord << ')' << std::endl;
                              exit(212);
                        }
                  }
                  else
                  {
                        std::cerr << "ERROR (there must be an '(') -> " <<
tokensVec[idx].value   <<   "   ("   <<   tokensVec[idx].x_coord   <<   ';'   <<
tokensVec[idx].y_coord << ')' << std::endl;
                        exit(213);
                  }
                  break;
            case LEX_BEGIN:
                  idx++;
                  start_begin();
                  while (tokensVec[idx].type == LEX_SEMICOLON)
                  {
                        idx++;
                        start_begin();
                  }
                  if(tokensVec[idx].type == LEX_END)
                  {
                        idx++;
                        break;
                  }
                  else
                  {
                        std::cerr  <<  "ERROR  (incorrect  input)  ->  "  <<
tokensVec[idx].value   <<   "   ("   <<   tokensVec[idx].x_coord   <<   ';'   <<
tokensVec[idx].y_coord << ')' << std::endl;
                        exit(214);
                  }
                  break;
case LEX_READLN:
                  idx++;
                  start_id(false);
                  while (tokensVec[idx].type == LEX_COMMA)
                  {
                        idx++;
                        start_id(false);
                  }
                  break;
            case LEX_WRITELN:
                  idx++;
                  start_V();
                  while (tokensVec[idx].type == LEX_COMMA){
                        idx++;
                        start_V();
                  }
                  break;
            case LEX_END:
                  if (idx == tokenCount - 1)
                        break;
                  else
                  {
std::cerr << "ERROR (incorrect input) -> " << tokensVec[idx].value << " (" <<
tokensVec[idx].x_coord << ';' << tokensVec[idx].y_coord << ')' << std::endl;
```

Продолжение листинга Б.1

```
                  exit(216);
            }
      }
      // Метод обработки выражения
      void start_V()
      {
            start_O();
            while (tokensVec[idx].type == LEX_EQ || tokensVec[idx].type ==
LEX_NEQ || tokensVec[idx].type == LEX_LSS
                  || tokensVec[idx].type == LEX_GTR || tokensVec[idx].type ==
LEX_REQ || tokensVec[idx].type == LEX_LEQ)
            {
                  ifWhileFlag = true;
                  idx++;
                  start_O();
            }
      }
      // Метод обработки операнда
      void start_O()
      {
            start_S();
            while (tokensVec[idx].type == LEX_PLUS || tokensVec[idx].type ==
LEX_MINUS || tokensVec[idx].type == LEX_OR)
            {
                  idx++;
                  start_S();
            }
      }
      // Метод обработки слагаемого
      void start_S()
      {
            start_M();
            while (tokensVec[idx].type == LEX_MULT || tokensVec[idx].type ==
LEX_DIV || tokensVec[idx].type == LEX_AND)
            {
                  idx++;
                  start_M();
            }
      }
// Метод обработки множителя
      void start_M()
      {
            if (tokensVec[idx].type == LEX_ID){
                  id_check();
                  idx++;
            }
            else if(tokensVec[idx].type == LEX_TRUE || tokensVec[idx].type ==
LEX_FALSE)
            {
                  ifWhileFlag = true;
                  idx++;
            }
            else if (tokensVec[idx].type == LEX_NUM)
            {
                  num_check();
                  idx++;
            }
            else if (tokensVec[idx].type == LEX_NOT)
            {
                  idx++;
                  start_M();
```

Продолжение листинга Б.1

```
}
            else if (tokensVec[idx].type == LEX_LPAREN)
            {
                  idx++;
                  if (tokensVec[idx].type == LEX_RPAREN)
                        idx++;
            }
            else
            {
                  std::cerr << "ERROR (there must be a variable or number) -> "
<< tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                  exit(217);
            }
      }
      // Метод проверяющий корректность введённого числа числа
      void num_check()
      {
            std::string str = tokensVec[idx].value;
            char maxS = 0;
            size_t fE = std::max(str.find('E'), str.find('e'));
            for (int i = 0; i < str.size() - 1; i++) maxS = std::max(maxS,
str[i]);
            if ((tolower(str[str.size() - 1]) == 'b' && maxS > 49) ||
(tolower(str[str.size() - 1]) == 'o' && maxS > 55)
            || (tolower(str[str.size() - 1]) == 'd' && maxS > 57) || ((fE <
std::string::npos) && !(maxS > 47 && 58 > maxS)))
            {
                  std::cerr << "ERROR (incorrect number) -> " <<
tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
                  exit(218);
            }
      }
      // Метод проверяющий объявлена ли переменная
      void id_check()
      {
            for (int i = 0; i < tokensVar.size(); i++)
            {
                  if (tokensVar[i].varName == tokensVec[idx].value)
                        return;
            }
            std::cerr << "ERROR (undeclared or incorrect variable) -> " <<
tokensVec[idx].value << " (" << tokensVec[idx].x_coord << ';' <<
tokensVec[idx].y_coord << ')' << std::endl;
            exit(219);
      }
};
```